APPENDIX A

```
//
//  ___________________________________________________________________
//  |-----------------------------------------------------------------|
//  |                                                                 |
//  |   FILE:   boolean.h                                             |
//  |   FUNCTIONALITY: Boolean definitions and max min                |
//  |   PROGRAM: required for all codes, checking for true/false for  |
//  |            readability                                          |
//  |   AUTHOR:  A. CHRISTIAN TAHAN                                   |
//  |   FIRST DRAFT:  02/10/00                                        |
//  |-----------------------------------------------------------------|
//  ___________________________________________________________________

#ifndef _Boolean_type

        #define _Boolean_type

        #define AND &&
        #define OR ||
        #define NOT !

        #ifdef BOOL
                #define Boolean BOOL
        #else
                  typedef int Boolean;
        #endif

        #ifndef FALSE
                #define FALSE 0
                #define TRUE 1
        #endif

#endif

#if !defined( __MINMAX_DEFINED)
#define __MINMAX_DEFINED
template <class T> T max(T x, T y)
        {
        return (x > y) ? x : y;
        };

template <class T> T min(T x, T y)
        {
        return (x < y) ? x : y;
        };
```

```
#endif
```

```
//
 //  ____________________________________________________________________
//  |-------------------------------------------------------------------|
//  |                                                                   |
//  |   FILE:   baseclas.cpp                                            |
//  |   FUNCTIONALITY: test program for memory allocation (baseclas)    |
//  |   PROGRAM:   server space program                                 |
//  |   COMMENTS:  basic structure for database interaction including   |
//  |              including file retrieval and bool program            |
//  |   AUTHOR: A. CHRISTIAN TAHAN                                      |
//  |   DATA FIRST VERSION: 02/10/00                                    |
//  |                                                                   |
//  |-------------------------------------------------------------------|
//  ____________________________________________________________________

#include "Baseclas.h"
#include "Baseclas.hpp"

class test
{

public:
  t_real ma;
  static t_real num;
#if (__BCPLUSPLUS__ ==  0x340)
  char car[8];
#else
  char car[3];
#endif

  //this operator is always required if the class must be used in
  //ImpObjectList
  test & operator=(const test & a)
  {
    ma=a.ma;
    car[0]=a.car[0];
    return ((*this));
  }
  Boolean operator==(const test & a)
  {
    return (ma==a.ma AND car[0]==a.car[0]AND car[1]==a.car[1]
            AND car[2]==a.car[2] );
  }
  test()
  {
    car[0]=car[1]=car[2]='a';
    ma=num;
    num++;
  }
  test(int val)
  {
    ma=val;
    return;
  };
  ~test()
  {
    car[0]=car[1]=car[2]='0';
    ma=0.0;
  }

};
t_real test::num=0;


Boolean Test_Base_Class(t_index max_num)
{
  t_index i;
  ImpObjectList<test> mat;
  test compare;
  mwarn<<"Base class Version " <<BASECLAS_VERSION;


  mwarn<<"testing base class allocating " <<max_num<<" elems of class
test";
  mat.Destroy_And_ReDim(max_num);

  mwarn<<"checking inizialization of " <<max_num<<" elems of class test";
  for (i=0;i<max_num;i++)
    {
      compare.ma=i+1;
      if (NOT (mat[i]==compare))
        {
          mwarn<<"Inizialization error on element: "<<i;
          return FALSE;
        }
    }

  mwarn<<"checking access for " <<max_num<<" elems of class test";
  for (i=0;i<max_num;i++)
    mat[i].ma=i+1;

  for (i=0;i<max_num;i++)
```

```
    if (mat[i].ma!=i+1)
      {
        mwarn<<"Inizialization error on element: "<<i;
        return FALSE;
      }
  {
    ImpSimpleTypeList<long> list1, list2 ;
    mwarn<<"check operator =";
    list1.Destroy_And_ReDim(70000UL);
    list1.Set(5);
    list2=list1;

    if (list2!=list1)
      {
        mwarn<<"operator = doesn't work on ImpSimpleList<long> ";
        return FALSE;
      }
  }
  mwarn<<"check complete";
  return TRUE;
}
```

```
//
//   ___________________________________________________________________
//   |-----------------------------------------------------------------|
//   |                                                                 |
//   |   FILE:   arraycla.cpp                                          |
//   |   FUNCTIONALITY: array                                          |
//   |   PROGRAM: required for all codes                               |
//   |   COMMENTS:  to arrange the database without taking too much    |
//   |              space                                              |
//   |   AUTHOR:  A. CHRISTIAN TAHAN                                   |
//   |   DATE FIRST VERSION: 02/16/00                                  |
//   |-----------------------------------------------------------------|
//   ___________________________________________________________________

#include "Arraycla.h"
#include "Arraycla.hpp"
#include "Diagclas.h"
#include "Diagclas.hpp"



MatrixOfDouble dummymat;

Boolean Test_Array_Mat_Functions()
        {
        MatrixOfDouble u,v,z;
        t_index dim=3,i,j;

        mwarn<<"Array class Version:"<<ARRAY_CLASS_VERSION;

        v.Destroy_And_ReDim(dim,dim);
        Check(v.Dim()==dim,
                 "Error in the instruction \"v.Destroy_And_ReDim(dim,dim)\"
v.Dim()="<<v.Dim());

        u.Destroy_And_ReDim(dim,dim);
        u=v;
        for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                        Check(u[i][j]==v[i][j],"Error in the instruction
\"u=v\"");

        u.Set(2);
        for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                        Check(u[i][j]==2.0,
                        "Error in the instruction \"u.Set(2)\" :u["<<i<<"]
["<<j<<"]="<<u[i][j]);

        z=u*2.0;
        for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                        Check(z[i][j]==4.0,
                        "Error in the instruction \"z=u*2.0\" :z["<<i<<"]
["<<j<<"]="<<z[i][j]);

         u/=2.0;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==1.0,
                 "Error in the instruction \"u/=2.0\" :u["<<i<<"]["<<j<<"]
="<<u[i][j]);

         u*=2.0;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==2.0,
                 "Error in the instruction \"u*=2.0\" :u["<<i<<"]["<<j<<"]
="<<u[i][j]);

         v=(u+2.0);
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(v[i][j]==4.0,
                 "Error in the instruction \"v=(u+2.0)\" :v["<<i<<"]
["<<j<<"]="<<v[i][j]);

         v=v/2.0;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(v[i][j]==2.0,
                 "Error in the instruction \"v=v/2.0\" :v["<<i<<"]["<<j<<"]
="<<v[i][j]);

         u+=(v+z-u);
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==6.0,
                 "Error in the instruction \"u+=(v+z-u)\" :u["<<i<<"]
["<<j<<"]="<<u[i][j]);

         u=u+v;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==8.0,
                 "Error in the instruction \"u=u+v\" :u["<<i<<"]["<<j<<"]
="<<u[i][j]);

         u-=2.0;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==6.0,
                 "Error in the instruction \"u-=2.0\" :u["<<i<<"]["<<j<<"]
="<<u[i][j]);

         u-=v;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==4.0,
                 "Error in the instruction \"u-=v\" :u["<<i<<"]["<<j<<"]
="<<u[i][j]);

         u=u|v;
         for (i=0;i<dim;i++)
                for (j=0;j<dim;j++)
                  Check(u[i][j]==24.0,
                 "Error in the instruction \"u=u|v\" :u["<<i<<"]["<<j<<"]
="<<u[i][j]);

         VetDouble vet;
         vet.Destroy_And_ReDim(dim);
         vet.Set(2);

         vet=u|vet;
     for (i=0;i<dim;i++)
           Check(vet[i]==144.0,
                 "Error in the instruction \"vet=u|vet\" :vet["<<i<<"]
="<<vet[i]);

         Check (u!=v, "Error in boolean function");

         u|=v;
     for (i=0;i<dim;i++)
            for(j=0;j<dim;j++)
               Check(u[i][j]==144.0,
                     "Error in the instruction \"u|=v\" :u["<<i<<"]
["<<j<<"]="<<u[i][j]);
```

```
MatrixOfDouble k, y, identity;
    t_real is_singular;
    t_index dim_k=4;

    identity.Destroy_And_ReDim(dim_k, dim_k);

    identity[0][0]=1.0;
    identity[0][1]=0.0;
    identity[0][2]=0.0;
    identity[0][3]=0.0;
    identity[1][0]=0.0;
    identity[1][1]=1.0;
    identity[1][2]=0.0;
identity[1][3]=0.0;
    identity[2][0]=0.0;
    identity[2][1]=0.0;
    identity[2][2]=1.0;
    identity[2][3]=0.0;
    identity[3][0]=0.0;
    identity[3][1]=0.0;
    identity[3][2]=0.0;
    identity[3][3]=1.0;


    k.Destroy_And_ReDim(dim_k, dim_k);
    y.Destroy_And_ReDim(dim_k, dim_k);

k[0][0]=4.0;
    k[0][1]=2.0;
k[0][2]=2.4;
    k[0][3]=2.0;
    k[1][0]=0.0;
    k[1][1]=8.0;
    k[1][2]=3.6;
    k[1][3]=8.7;
    k[2][0]=5.1;
    k[2][1]=9.3;
    k[2][2]=2.9;
    k[2][3]=3.1;
    k[3][0]=7.23;
    k[3][1]=5.7;
    k[3][2]=1.9;
    k[3][3]=4.98;

    y=k;
```

```
         is_singular=k.Inverse();

         Check(is_singular!=0.0,"Routine Inverse() doesn't work,
is_singular="<<is_singular);

         k=y|k;

         k.Chop();

         Check(k==identity,"Routine Inverse() doesn't work");

         t_index dim_y=2;
     VetDouble vect;

         y.Destroy_And_ReDim(dim_y, dim_y);

         y[0][0]=2;
         y[0][1]=1;
         y[1][0]=1;
         y[1][1]=2;

         y.EigenValues_And_EigenVectors(vect, k);

         MatrixOfDouble eigval,tr,res;
         eigval.Destroy_And_ReDim(dim_y,dim_y);

         for (i=0;i<dim_y;i++)
           for (j=0;j<dim_y;j++)
                  if (i==j)
            eigval[i][j]=vect[i];
                  else eigval[i][j]=0.0;


         tr.Transpose_Of(k);

         res=tr|eigval|k;

         res*=res;
         y*=y;
         k=res-y;
         k.Chop();

         for (i=0;i<dim_y;i++)
           for (j=0;j<dim_y;j++)
                 Check(k[i][j]<=PRECISION,"routines
EigenValues_And_EigenVectors don't work");
```

```
         DiagMatrixOfDouble diag;
         MatrixOfDouble kk;
     t_index dim_diag=4;

     diag.Destroy_And_ReDim(dim_diag, dim_diag);

     diag[0][0]=4.0;
         diag[1][1]=6.0;
         diag[2][2]=5.1;
         diag[3][3]=7.23;

         k<<=diag;

         y.Destroy_And_ReDim(dim_diag, dim_diag);

     y[0][0]=4.0;
         y[0][1]=0.0;
     y[0][2]=0.0;
         y[0][3]=0.0;
         y[1][0]=0.0;
         y[1][1]=6.0;
         y[1][2]=0.0;
         y[1][3]=0.0;
         y[2][0]=0.0;
         y[2][1]=0.0;
         y[2][2]=5.1;
         y[2][3]=0.0;
         y[3][0]=0.0;
         y[3][1]=0.0;
         y[3][2]=0.0;
         y[3][3]=7.23;

         Check(y==k, "Routine Change_Diag2Full don't work");

         return TRUE;
        }


// Housholder method transforms a symmetric matrix into a tridiagonal one
void MatrixOfDouble::Householder()
        {
        VetDouble x, y, u;
        MatrixOfDouble H, U, UT, I;

        t_index dim, i, k;
```

```
t_real sum;

dim=(*this)[0].Dim();
// B is a symmetric matrix of order greater than two
// the symmetric property is not tested
Assert(dim>2);

x.Destroy_And_ReDim(dim);
y.Destroy_And_ReDim(dim);
H.Destroy_And_ReDim(dim,dim);
U.Destroy_And_ReDim(dim,dim);
UT.Destroy_And_ReDim(dim,dim);
I.Destroy_And_ReDim(dim,dim);

for(i=0;i<dim;i++)
        I[i][i]=1.0;

for(k=0;k<dim-2;k++)
        {
        UT.Set(0);
        H.Set(0);
        y.Set(0);

        for(i=0;i<dim;i++)
                x[i]=(*this)[i][k];

        for(i=0;i<=k;i++)
                y[i]=x[i];

        sum=0;
        for(i=k+1;i<dim;i++)
                sum+=pow(x[i],2);
        sum=sqrt(sum);
        if(x[k+1]>0)
                sum=-sum;

        y[k+1]=sum;
        UT[0]=x-y;

        // calculate the x-y norm
        sum=0;
        for(i=0;i<dim;i++)
                sum+=pow(UT[0][i],2);
        sum=sqrt(sum);

        UT[0]/=sum;
```

```
                U.Transpose_Of(UT);

                U=U|UT*2;
                H=I-U;
                (*this)=H|(*this)|H;
                }

        return;
        }


#define SIGN(a,b) ((b)<0 ? -fabs(a) : fabs(a))

void MatrixOfDouble::EigenValues_And_EigenVectors(VetDouble& eigenvalues,
                                                         MatrixOfDouble&
eigenvectors) const
        {
        t_signed i, l;
        t_index m, j, k, iter, dim;
        t_real s, r, p, g, f, dd, c, b;
        VetDouble external, diagonal, eig_values;
        MatrixOfDouble tridiagonal;

        tridiagonal = (*this);

        dim= (*this).Dim();
        eigenvalues.Destroy_And_ReDim(dim);
        external.Destroy_And_ReDim(dim);
        diagonal.Destroy_And_ReDim(dim);
        eigenvectors.Destroy_And_ReDim(dim,dim);

        eigenvectors.Set(0.0);

        tridiagonal.Householder();

        for(i=1;i<(t_signed)dim;i++)
                {
                external[i]=tridiagonal[i-1][i];
                diagonal[i]=tridiagonal[i][i];
                eigenvectors[i][i]=1.0;
                }

        for(i=1;i<(t_signed)dim;i++)
                external[i-1]=external[i];

        external[dim-1]=0.0;
        for (l=0;l<(t_signed)dim;l++)
                {
                iter=0;
                do {
                        for (m=l;m<dim-1;m++)
                                {
                                dd=fabs(diagonal[m])+fabs(diagonal[m+1]);
                                if (fabs(external[m])+dd == dd)
                                        break;
                                }
                        if ((t_signed)m != l)
                                {
                                if (iter++ == 30)
                                        cout<<"Too many iterations";
                                g=(diagonal[l+1]-diagonal[l])/(2.0*external
[l]);

                                r=sqrt((g*g)+1.0);
                                g=diagonal[m]-diagonal[l]+external[l]/(g
+SIGN(r,g));

                                s=c=1.0;
                                p=0.0;
                                for (i=m-1;i>=l;i--)
                                        {
                                        f=s*external[i];
                                        b=c*external[i];
                                        if (fabs(f) >= fabs(g))
                                                {
                                                c=g/f;
                                                r=sqrt((c*c)+1.0);
                                                external[i+1]=f*r;
                                                c *= (s=1.0/r);
                                                }
                                        else
                                                {
                                                s=f/g;
                                                r=sqrt((s*s)+1.0);
                                                external[i+1]=g*r;
                                                s *= (c=1.0/r);
                                                }
                                        g=diagonal[i+1]-p;
                                        r=(diagonal[i]-g)*s+2.0*c*b;
                                        p=s*r;
                                        diagonal[i+1]=g+p;
                                        g=c*r-b;
                                        for (k=0;k<dim;k++)
```

```
                                                {
                                                f=eigenvectors[k][i+1];
                                                eigenvectors[k][i+1]
=s*eigenvectors[k][i]+c*f;
                                                eigenvectors[k][i]
=c*eigenvectors[k][i]-s*f;
                                                }
                                        }
                                diagonal[l]=diagonal[l]-p;
                                external[l]=g;
                                external[m]=0.0;
                                }
                        } while ((t_signed)m != l);
                }

        // transposition of eigenvectors matrix in order to have
autovectors on the rows
        // and not on the columns
        t_real temp;
        for(i=0;i<(t_signed)dim-1;i++)
                 for(j=i+1;j<=dim-1;j++)
                        {
                        temp = eigenvectors[i][j];
                        eigenvectors[i][j] = eigenvectors[j][i];
                        eigenvectors[j][i] = temp;
                        }
        return;
        };


// *******************************************************************   *
// *                                                                     *
//                      Matrix of double                                 *
// *                                                                     *
// *******************************************************************   *


// Perform a low up decomposition from a square matrix
void MatrixOfDouble::Low_Up_Dcmp(MatrixOfDouble &L, MatrixOfDouble &U)
        {
        t_index i,j,k,n;
        t_real sum;

        Assert((*this).Dim_Row()==(*this).Dim_Col());

        n=(*this).Dim_Row();
        L.Destroy_And_ReDim(n,n);
        U.Destroy_And_ReDim(n,n);

        for(j=0;j<n;j++)
                L[j][j]=1.0;

        j=0;
        for(i=0;i<n;i++)
                U[j][i]=(*this)[j][i];
        for(i=1;i<n;i++)
                L[i][j]=(*this)[i][j]/U[j][j];

        for(j=1;j<n;j++)
                {
                for(i=j;i<n;i++)
                        {
                        sum=0.0;
                        for(k=0;k<=j-1;k++)
                                sum+=L[j][k]*U[k][i];
                        U[j][i]=(*this)[j][i]-sum;
                        }

                for(i=j+1;i<n;i++)
                        {
                        sum=0.0;
                        for(k=0;k<=j-1;k++)
                                sum+=L[i][k]*U[k][j];
                        L[i][j]=((*this)[i][j]-sum)/U[j][j];
                        }
                }

        return;
        }

// Solve a linear equation Ax=b where A is a lower triangular or
// upper triangular matrix
void MatrixOfDouble::Solve_Triangular(VetDouble &y, VetDouble &b)
        {
        t_index i,k,n;
        t_real sum;

        n=b.Dim();

        //Assert(tri_mat.Dim_Row()==tri_mat.Dim_Col());
        Assert(n==(*this).Dim_Row());
        y.Destroy_And_ReDim(n);

        if((*this)[0][1]==0.0)
                {
                y[0]=b[0];
                for(i=1;i<n;i++)
                        {
                        sum=0.0;
                        for(k=0;k<=i-1;k++)
                                sum+=(*this)[i][k]*y[k];

                        y[i]=b[i]-sum;
                        }
                }
        else{
                y[n-1]=b[n-1]/(*this)[n-1][n-1];
                for(i=n-1;i>0;i--)
                        {
                        sum=0.0;
                        for(k=n-1;k>=i;k--)
                                sum+=(*this)[i-1][k]*y[k];

                        y[i-1]=(b[i-1]-sum)/(*this)[i-1][i-1];
                        }
                }

        return;
        }

void MatrixOfDouble::Transpose_Of(const MatrixOfDouble & x)
        {
        Destroy_And_ReDim(x.Dim_Col(),x.Dim_Row());
        t_index i,j;

        for(i=0;i<x.Dim_Row();i++)
                for(j=0;j<x.Dim_Col();j++)
                        (*this)[j][i]=x[i][j];

        return;
        }

t_real MatrixOfDouble::Inverse()
        {
        MatrixOfDouble L,U,B,X;
        VetDouble y;
        t_index N,k;
```

```
t_real det;

N=(*this).Dim_Col();
Assert((*this).Dim_Row()==N);

(*this).Low_Up_Dcmp(L, U);

B.Destroy_And_ReDim(N,N);
X.Destroy_And_ReDim(N,N);

det=1.0;
for(k=0;k<N;k++)
        {
        B[k][k]=1.0;
        det*=U[k][k];
        }

if(fabs(det)<=10E-10)
        mwarn<<"Singular matrix in low up decomposition";

for(k=0;k<N;k++)
        {
        L.Solve_Triangular(y, B[k]);
        U.Solve_Triangular(X[k], y);
        }

(*this).Transpose_Of(X);

return det;
}
```

```
//
//
//  ____________________________________________________________________
//  |------------------------------------------------------------------|
//  |                                                                  |
//  |   FILE:   diagclas.cpp                                           |
//  |   FUNCTIONALITY: diagonal matrix                                 |
//  |   PROGRAM: required to  all codes                                |
//  |   COMMENTS:header template classes, for access to header files   |
//  |   AUTHOR: A. CHRISTIAN TAHAN                                     |
//  |   DATE FIRST VERSION:  02/13/00                                  |
//  |------------------------------------------------------------------|
//  ____________________________________________________________________

#include "Diagnost.h"
#include "Diagclas.hpp"
#include "Arraycla.h"
#include "Arraycla.hpp"

Boolean Test_Math_Diagonal_Matrix_Function ()
{
  DiagMatrixOfDouble u,v,z,c;

  t_index dim=10, i;

  mwarn<<"diagclas version"<<DIAG_CLASS_VERSION;

  u.Destroy_And_ReDim(dim,dim);
  v.Destroy_And_ReDim(dim,dim);
  z.Destroy_And_ReDim(dim,dim);
  c.Destroy_And_ReDim(dim,dim);

  Check (v.Dim()==dim, "Error in Destroy_And_ReDim() function");

  u.Set(2);
  v.Set(3);
  z.Set(2);
  z*=u*v*u*u*5;

  c.Set(240);

  Check(c==z,"diagonal matrix routines for the product don't work");

  c+=u+80+v;
  z.Set(325);
  Check(c==z,"diagonal matrix routines for the addition don't work");
  Check(z[dim-1][dim-1]==c[0][0],"diagonal matrix routine operator[] don't
work");

  c/=25;
  for (i=0;i<dim;i++)
    Check(c[i][i]==13.0,"diagonal routines operator/= don't work");

  c=u|v|v;
  for (i=0;i<dim;i++)
    Check(c[i][i]==18.0,"diagonal routines operator| don't work");

  c=u|v;
  for (i=0;i<dim;i++)
    Check(c[i][i]==6.0,"diagonal routines operator| don't work");

  VetDouble vect, ris;

  vect.Destroy_And_ReDim(dim);
  vect.Set(2);

  ris=c|vect;
  for (i=0;i<dim;i++)
    Check(ris[i]==12.0,"diagonal routines operator| don't work");

  v.Set(3);

  c|=v;
  for (i=0;i<dim;i++)
    Check(c[i][i]==18.0,"diagonal routines operator|= don't work");
  u[0][0]=12.0;
  u[1][1]=22.5;
  u[2][2]=343.1;
  u[3][3]=125.4;
  u[4][4]=2.0;
  u[5][5]=458.1;
  u[6][6]=75.2;
  u[7][7]=45.126;
  u[8][8]=75.2;
  u[9][9]=45.3;

  v=u;
  u.Inverse();
  c=v|u;
  c.Chop();
  for(i=0;i<dim;i++)
    Check(c[i][i]==1.0,
          "diagonal matrix routines Inverse() don't work : c["<<i<<"]="<<c
[i][i]);

  DiagMatrixOfDouble eigvect;

  v.Set(3);
  v.EigenValues_And_EigenVectors(vect,eigvect);

  for(i=0;i<dim;i++)
    u[i][i]=vect[i];

  Check(u==v,"error in function EigenValues_And_EigenVectors");

  return TRUE;
}
```